Keeping Secrets: Multi-objective Genetic Improvement for Detecting and Reducing Information Leakage

Ibrahim Mesecan
Iowa State University
Ames, Iowa, USA
imesecan@iastate.edu

Daniel Blackwell
London, UK
daniel.blackwell.14@ucl.ac.uk

David Clark
London, UK
david.clark@ucl.ac.uk

Myra B. Cohen
Iowa State University
Ames, Iowa, USA
mcohen@iastate.edu

Justyna Petke
London, UK
j.petke@ucl.ac.uk

ABSTRACT

Information leaks in software can unintentionally reveal private data, yet they are hard to detect and fix. Although several methods have been proposed to detect leakage, such as static verification-based approaches, they require specialist knowledge, and are time-consuming. Recently, HyperGI introduced a dynamic, hypertest-based approach that detects and produces potential fixes for information leakage. Its fitness function tries to balance information leakage and program correctness, but as the authors of that work point out, there may be a tradeoff between keeping program semantics and reducing information leakage.

In this work we ask if it is possible to automatically detect and repair information leakage in more realistic programs without requiring specialist knowledge. Our approach, called LeakReducer explicitly encodes the tradeoff between program correctness and information leakage as a multi-objective optimisation problem. We apply LeakReducer to a set of leaky programs including the well known Heartbleed bug. It is comparable with HyperGI on their toy applications. In addition, we demonstrate it can find and reduce leakage in real applications and we see diverse solutions on our Pareto front. Upon investigation we find that having a Pareto front helps with some types of information leakage, but not all.

CCS CONCEPTS

• **Software and its engineering → Search-based software engineering**; **Search-based software engineering**; • **Security and privacy → Software security engineering**.

KEYWORDS

Genetic Improvement, Information Leakage, Search-Based Software Engineering, Automated Program Repair

ACM Reference Format:
Ibrahim Mesecan, Daniel Blackwell, David Clark, Myra B. Cohen, and Justyna Petke. 2022. Keeping Secrets: Multi-objective Genetic Improvement for Detecting and Reducing Information Leakage. In *Proceedings of The 37th IEEE/ACM International Conference on Automated Software Engineering (ASE 2022).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Information leakage from programs has led to high profile security bugs such as the Heartbleed bug.[1] Typically, information leaks from a program when either it contains information flow control (IFC) errors or when a data structure such as a buffer or stack can be accessed in an unbounded manner. The Heartbleed bug was due to a problem in the OpenSSL cryptographic library. When pinged with a malformed query, it was possible to read past the buffer and return unencrypted data from the server's memory, hence providing a backdoor for eavesdropping on network traffic. This bug existed for several years in a library used by programs and servers worldwide. While it represents a common type of information leakage, more subtle IFC leaks can occur, and these may also lead to exposure of private information.

Take, for instance, the program below which accepts an integer variable `var` and returns 0, 1 or 2. It has two predicates, one which compares var against a variable called `magic_number` and one that uses the value of a secret variable (`protected_var`).

```
int leaking_secrets(int var){
   ...
   if(var > magic_number}{
       leak_info=2;
   }
   else if(var < protected_var){
       leak_info=1;
   }
   else
       leak_info=0;
   ...
   return leak_info; }
```

An important part of information leakage is setting a security policy (beyond the scope of this work). Based on a given security policy, we can assume that 1) `var` and `magic_number` are publicly known (a.k.a low security variables), and 2) `protected_var` is considered high security; it is a secret. Based on this policy, it is not hard to see that repeated querying of this program with different

[1]https://heartbleed.com/

values of `var` can expose information about `protected_var`. Let's assume `magic_var=6` and `protected_var=5`. If we run the program repeatedly using the inputs 8, 3, 5, we get different return values of 2, 1, 0. If we then evaluate this using numbers 0, 1, 2, we get return values of 1, 1, 1. This gives us information about the content of `protected_var`, i.e. information is leaking. As we can see, the value of `magic_number` could impact the visibility of this information. Even in this simple program the ability to discover the leak dynamically depends on multiple variables (and control flows). If, for instance, `magic_number` is set to a large negative value, then for most inputs of `var` no information about `protected_var` will be revealed. The program always returns 2.

Finding information leakage in programs is non-trivial, with the most common approach being static analysis [23, 39]. There have also been some combined static/dynamic techniques [34], but these have not necessarily been applied to real-world programs nor can they provide patch suggestions if the problem is found. Moreover, as we show in our study, current techniques may be able to detect information leakage related to memory overflows, but they may still miss those related to a program's control flow as is exemplified in the example program `leaking_secrets`. Mechtaev et al. [29] were able to use automated program repair to fix the Heartbleed bug suggesting automated ways to handle these types of faults. However, that work assumes the leakage would break program functionality and requires failing test cases. Following earlier work [31], we call such tests *functional tests*. Information leakage can occur even if a program passes all functional tests, making information leakage difficult to detect and fix.

More recently, Mesecan et al. [31] presented an approach called HyperGI which uses hypertests and genetic improvement to first detect and then repair information leakage in programs. While this work makes some advances in reducing information leakage it has several drawbacks. We contacted the authors for artifacts and learned that: (1) HyperGI was applied only to small functions (less than 40 lines of code) (2) and results obtained were not from a fully automated setting, as user had to insert some domain knowledge (i.e., extra variables with their types) to be used during search.

A key finding the authors of HyperGI point out in their paper, is that the patches produced present a tradeoff between preserving original functionality (as exposed by functional tests) and reducing information leakage (as exposed by hypertests). Returning to our example, the only way to reduce information leakage completely, would be to change the predicate related to `protected_var`. However, that changes the initial intended program and some functional tests are likely to fail. This suggests repairing information flow leakage should be viewed as a multi-objective problem. We should consider the option of balancing the leakage of secrets with the need for particular program behavior.

In this paper, we present an automated multi-objective framework for estimating and reducing information leakage. It requires only a program and its security policy. We implement our approach in LeakReducer. Moreover, we improve upon HyperGI by: (1) using multiple functional test sets as input; (2) using automatically derived repair ingredients; (3) including both single and multi-objective search strategies; and (4) scaling to real programs. We have evaluated LeakReducer on the prior subjects from HyperGI as well as on three other programs, including two modules from the `OpenSSL` library. We were able to both detect and reduce leakage in two files, each around 1,000 lines of code, including the original Heartbleed bug. One of our detected leaks in a real-world program turned out to be a false positive (confirmed by developers), but it points to a different use case and class of information leakage faults which we aim to study as future work. Furthermore, we examine the quality of Pareto fronts for different multi-objective algorithms, and explore a few interesting patches in depth.

In summary, the contributions of this work are:

1. An automated multi-objective approach, LeakReducer, for finding and reducing information leakage;
2. An empirical study demonstrating the effectiveness of LeakReducer; and
3. An evaluation of patch diversity produced by LeakReducer, detailing tradeoffs between change in original semantics of the given program and reduction in information leakage.

2 BACKGROUND AND RELATED WORK

The information flow control problem of maintaining data confidentiality across program executions has a long history, part of which was surveyed in the early 2000s by Sabelfeld and Myers [38]. Software should be designed so that it obeys a security flow policy, or *noninterference* policy: low security users should not be aware of the actions of high security users [20]. As in this paper, security policies are often expressed in terms of two classes of user, high and low, although it is allowable that they be arbitrarily complex as in Dorothy Denning's lattice model [17].

We can partition data, e.g. a memory state in an imperative program, using High/Low labels, then use the partition to formally define the noninterference property for the program and security policy pair. We say that a pair of states, s_1, s_2, are low equivalent, $s_1 \equiv_L s_2$, if the parts of the states labelled Low are the same.

Definition 1 (Noninterference property). *A program P satisfies the noninterference property for the High-Low security policy if for every pair of initial states* s_1, s_2, $s_1 \equiv_L s_2 \Rightarrow P(s_1) \equiv_L P(s_2)$.

In this flow and termination insensitive definition, noninterference means the program maps all low equivalent initial state pairs to low equivalent final state pairs.

Noninterference is not a property of single executions but of *pairs* of executions. Clarkson and Schneider generalised this idea, calling such program properties *hyperproperties* as they are only expressible using sets of sets of executions [14]. The "preservation of low equivalence" property partitions all distinct pairs of executions that begin in low equivalent states into the set of pairs that preserve the low equivalence in the final states and the set of pairs that do not. The noninterference property says that the latter set is empty. If it is not, information leaks from the High labelled parts of initial states to the Low labelled parts of final states. This last observation leads to what Kinder called *Hypertests* [24], using pairs of low equivalent inputs with differing high inputs with a "built in" oracle that fails the hypertest if the outputs are not low equivalent. This is the method we use to detect information leaks in this paper. To repair information leaks, we need to estimate leak size.

2.1 Quantified Information Flow

Historically, much of the research focus on how to *check* that code obeys its designated security policy has been heavily influenced by

Volpano, Irvine, and Smith's work on security type systems [48]. This has led to tools such as the Jif compiler and IDE [7]. One difficulty lies not in the type system approach itself, but in the noninterference property, recognised to be overly restrictive for real programs – for example, a password checking program famously will not satisfy the property for a security policy that protects the correct password. Attempts to ameliorate this restrictiveness include adding declassification to type systems [11] and Secure Multi Execution of programs which guarantees that low users cannot learn confidential information no matter whether the program satisfies the security policy or not [37]. Relevant to this paper is research in Quantified Information Flow (QIF) using information theory which had the original aim that quantitative security policies could allow information to leak, but only in a bounded way [4, 12]. While it was eventually realised that bounding QIF or exact calculation of QIF for programs is, in the worst case, computationally feasible though intractable (see Terauchi and others [51]), the attractiveness of the idea means research continues on approaches to the bounding problem [10].

We, however, aim to detect the *existence* of a leak via hypertesting, localize its cause in the code, then use Genetic Improvement (GI) to eliminate or reduce the leak size. The localization and elimination steps rely on QIF estimates but we do not need to provide guarantees for the leak bound. While it is true that eliminating leaks completely is provably equivalent to satisfying noninterference [13], as a test-based methodology, we do not guarantee the absence of leaks, only that we *may* find and fix them.

We follow Mesecan et al. [31] in using the conditional Shannon entropy of Low outputs given Low inputs as the measure of leakage, then estimating this using test sets that we assume are sampled from a discrete uniform distribution. Ultimately, the fundamental definitions were provided by Shannon [16], and the leakage definition by Clark, Hunt and Malacaria [13]. In what follows we present two leakage definitions and comment on how they relate to programs and security policies.

Definition 2 (Leakage measure for deterministic programs). *Let H and H' be the random variables in the high parts of the initial and final states of program P, respectively while L and L' are the corresponding random variables in the low parts. Then if P is deterministic program and high and low parts partition each state, $\mathcal{L}(P)$, the leakage from H into L' for P is given by $\mathcal{L}(P) = \mathcal{H}(L'|L)$ where $\mathcal{H}$ is the Shannon entropy of a random variable.*

This measures the information that flows from a random variable in the high part of the initial states to another random variable in the low part of the final states, on the following assumptions: (1) we account for all contributions to the initial states, (2) these are partitioned between high and low, and (3) the program is deterministic; once we account for the low part of the initial states any remaining entropy in the low part of the final states must be due to the high part of the initial states.

The advantage in this specialized definition is that you do not need to know anything about the random variable in the high part of the initial states. But its underlying assumptions break if some other source of information correlates with the low part of the final state during execution. In particular, use of the definition in a test-based scenario can be sensitive to external and internal nondeterminism during executions, e.g. changing configuration parameters or race conditions in threads. In the presence of noise, Clark et al. recommend using the more general definition, $\mathcal{L}(P) = \mathcal{I}(H; L'|L)$, and show that the two definitions are equivalent under the assumptions [13].

In our method, because we don't need a precise bound on leakage, some flakiness in hypertests is tolerable. The tradeoffs inherent in multi-objective solutions will not always include reducing the leakage to zero. Also, using a definition that is agnostic about the entropy in the high part of states allows us to perform *experiments* where control and observation is partial as opposed to true testing where control and observation is complete. A common security policy for Unix utilities is to label inputs and outputs low and designate data in the memory of the process as high [27].

While much software is open source, security policies tend to be implicit at best and certainly not open source. We thus re-use ones from previous work [31], outlining others in Section 4.1.

2.2 Related work

Mesecan et al. [31] were the first to use hyper-testing and genetic improvement to detect and fix information leaks in software. They demonstrated the initial promise of this approach on a small set of programs. However, by using a single-objective approach, their method necessarily folded tradeoff decisions between leakage and functionality into their chosen fitness function. We believe that this tradeoff should be available to developers and consequently that leakage repair should be based on multi-objective search.

In order to achieve good repair results Mesecan et al. reported user input was necessary. In particular, the user had to add ingredients (e.g., variables) to the search engine to help find a repair. We have automated all the steps of that previous work. All the user is required to provide is a program and its security policy to apply our approach.

Some interest in hypertesting programs for information leaks has been evident in the fuzzing research community though, rather than focus on semantic leaks as we do, the interest has been in side channel leaks. CT-fuzz [21] and QFuzz [35] are examples of recent research in this area. Neither deals with automating repair. CT-Fuzz looks for failing hypertests as evidence of leakage. Its oracle is coarse, with an observation power limited to path divergence but including timing differences. QFuzz uses a leakage measure based on min entropy to analyse the size of leaks from timing channels. Our work relates to that of the automated program repair community but our methodology is very different. There does exist work on automated program repair in the side channel leakage community, for example Athanasiou et alia's work [8]. This work ultimately derives from Agat's work on masking timing channels [3] but is more sophisticated, exploiting creation of statistical independence in the representation of secret data to do the masking.

3 LEAKREDUCER

We now present our framework for multi-objective information leakage reduction, LeakReducer. Figure 1 shows an overview.

First, a user needs to provide a program and a security policy (stage (a) in Figure 1). Next, LeakReducer requires two test sets: (1) a hypertest set, described in Section 3.2, stage (b) in Figure 1; and (2) a functional test set, described in Section 3.1, stage (c) in

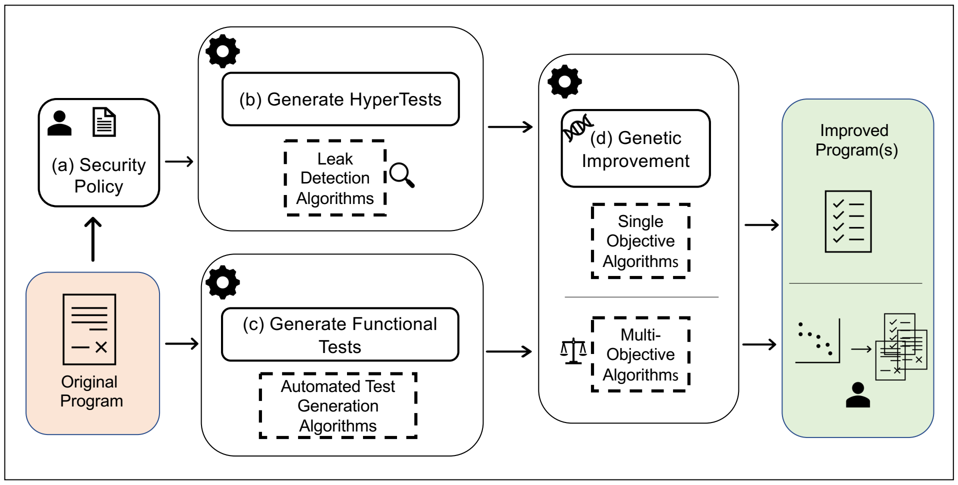
(a) Security Policy
Original Program
(b) Generate HyperTests
Leak Detection Algorithms
(c) Generate Functional Tests
Automated Test Generation Algorithms
(d) Genetic Improvement
Single Objective Algorithms
Multi-Objective Algorithms
Improved Program(s)

Figure 1: Overview of LeakReducer. The starting point is a potentially leaky program. First a security policy is used to generate hypertests for leak detection. Automated test generation is used for functional tests. Using these test suites the program is improved using either a single or multi-objective algorithm. The result is either a single program or a Pareto front (PF) of programs from which the developer can choose.

Figure 1. The functional test set is used to check whether current functionality is preserved; and the hypertest set is used to quantify information leakage. Assuming the hypertests discover a non-zero information leak, then, the repair stage, stage (d) in Figure 1, can start. This stage follows the usual genetic improvement process (Section 3.3): localization of the most promising parts of program for optimization (in our case leak reduction, described in Section 3.3.2); generation of candidate patches, (Section 3.3.3); and search over the generated patches guided by a fitness, (Section 3.3.4). Finally, we discuss implementation details in Section 3.4.

3.1 Automated Functional Test Generation

We automatically generate test sets that capture the current functionality of the given software. Any existing test suite could be used, but such tests are not always available. For instance, no tests covered the faulty function in OpenSSL before the Heartbleed bug was discovered. We always assume a real-world scenario where it's simply unknown whether a bug exists or not and discover any problems using hypertests, in contradiction to, say, Mechtaev et al. [29] where their automatic repair of the HeartBleed bug was reliant on test cases only added *after* the bug was discovered.

There is a plethora of automated test generation techniques to choose from [6, 19, 49]. We decided to use fuzzers due to their increasing popularity both in academia and in industry [32], effectiveness at finding software vulnerabilities [28] and programming language independence. We have, however, introduced two improvements described below.

Intuitively, normal use of a given program is unlikely to trigger information leaks (assuming the developers did their due diligence). It is the rare, perhaps malicious, inputs that are likely to reveal such faults. Grey-box fuzzers generate inputs, driven by the goal of maximizing code coverage, and might sometimes miss such rare events. Therefore, we use a program transformation technique, HashFuzz [30], in conjunction with a fuzzer to increase the diversity of generated inputs for one of our test generation techniques. Given the rare nature of leakage triggering paths, and inability to rely on coverage feedback for discovering leaks, we felt that increased input diversity could be key to automated discovery of future leaks.

Moreover, we noticed that even with HashFuzz certain branches can be missed. As an example, Atalk [22] (one of our subject programs) has a branch that is only reachable by matching a 32-bit variable with a specific constant: `TCP_ ESTABLISHED`. If the fuzzer was only generating values for this particular 32-bit variable, then the probability of discovering this branch for each generated input would be just $1/2^{32}$. To further increase code coverage we allow for the user to provide input seeds as a basis to begin fuzzing.

3.2 Automated Hypertest Generation

As shown in previous work [31], traditional tests are not always able to reveal information leaks. Therefore, *hypertests* are needed to detect and measure the amount of leakage from the target program. We use the same strategy as Mesecan et al. [31]. In particular, given a security policy, LeakReducer conducts a binary search on the low variable values. A random value is selected from each half of the low value range in bits. To create hypertests, for each selected value a similar search is conducted on the high value range. The process is repeated until we reach a fixed number of low and high values. If no leak is found, we repeat the process with a different budget. The search spaces can be huge, hence we restrict the number of values (and thus hypertests) we sample during this process.

3.3 Genetic Improvement

Once LeakReducer has generated functional (Section 3.1) and hypertest (Section 3.2) test sets, we can define the fitness functions which will guide the search algorithms in the leak reduction stage (stage (d) in Figure 1). This stage utilizes Genetic Improvement (GI) [36] to find improved program versions. The following subsections describe each of the steps of the process.

3.3.1 Fitness Function. The aim of LeakReducer is to find a patch that will reduce information leakage, whilst preserving software functionality. Since the two objectives can be in conflict with each other (albeit not always), we need to quantify them separately.

The first objective is quantified by the pass rate of the functional tests. These should all pass when run on the unmodified program, as they are used as a proxy for the intended program behaviour.

The second objective is quantified by using a hypertest set to estimate the leak size. We use the same calculation as HyperGI [31] (see Definition 2 in Section 2), and henceforth use QIF to refer to estimate for quantified information leakage.

3.3.2 Leak Localization. Before starting the search process, possible leak locations need to be identified. Similar to previous work [31], we use a lightweight form of dynamic analysis for this purpose, albeit on the whole leaky file not just the function. Based on the security policy, we first identify the file where leakage occurs. Next, we remove each statement from the target, one-by-one, and observe the impact on QIF when the hypertests are run. When a statement is removed, if the program fails to compile or run, then QIF for that statement is assumed to be zero. Otherwise, we calculate and store the absolute change in QIF of this modified program from the original. Finally, statement leakages are normalized with respect to the highest absolute difference. The higher the absolute difference, the greater the probability that the fault is coupled with that statement. We use these probabilities during the search process to prioritize statements that have higher impact on information leakage based on this analysis. The assumption is that changes to those statements should have highest impact on (and hopefully reduce) the leakage.

3.3.3 Patch Generation. Once we identified statements that have influence on information leakage, we mutate them to create new program variants. In genetic improvement the standard mutation operators simply insert, delete and replace software fragments, e.g., code statements. We observe that information leakage problems are caused by information flow between low and high variables in a given program. Therefore, aside from traditional GI operators, we use the *NewIf* and *NewFor* mutation operators, introduced in Mesecan et al. [31]. In LeakReducer, we fully automate this step of the process. In order to create a new statement, we need expressions to populate the parentheses for FOR and IF statements. To reduce the search space, we construct comparison expressions using existing identifiers and type match them, so that, e.g., a number is compared with another number. In Mesecan et al. [31] the user was prompted to provide such information. For both *NewIf* and *NewFor*, the body statement to be executed is selected from existing code. The *NewIf* comparison operator is selected at random.

3.3.4 Search Strategies. Another improvement we introduce over HyperGI is the option to choose between multiple search strategies, including multi-objective search. We aim to reduce both 1) the number of failing test cases, i.e., *fail_rate* and 2) information flow leakage, measured with *QIF*. The optimal solution will reduce both of these quantities to 0.

3.3.5 Single Objective Optimization. The single-objective search option uses the same fitness function as HyperGI [31].

$$fitness = (fail\_rate + \frac{QIF_i}{QIF_0})/2 \qquad (1)$$

As can be seen from equation 1, this fitness function balances the two objectives: *fail_rate* and normalized QIF where QIF_0 is the initial leakage ($QIF_0 > 0$), and QIF_i is the leakage of current program variant ($QIF_i \geq 0$).

3.3.6 Multi-objective Optimization. In multi-objective optimization, there are several objectives where maximizing (or minimizing) one objective may have conflicting results with other objectives. LeakReducer has two objectives: reducing the information leakage and preserving intended functionality.

LeakReducer's multi-objective [15] setting thus reports a list of non-dominated Pareto front solutions, where every solution in the list is better than the other solutions in at least one objective. This provides the decision maker with more options to choose from.

3.4 Implementation Details

In order to implement LeakReducer, we extended an existing genetic improvement framework, PyGGI [5] and integrated it with the JMetalPy framework for multi-objective optimization [9]. This integration was a non-trivial task, as it required integration of frameworks with different architectures. PyGGI provides us with the GI framework, while JMetalPy provides different multi-objective algorithms and Pareto front quality metrics.

For functional test case generation, we investigated several state-of-the-art fuzzing strategies (Section 3.1). We use AFL++ 3.12 [18] which consistently ranks amongst the best fuzzers in terms of program exploration ability [33]. The HashFuzz transformation was also applied to the tested programs, and these were fuzzed using the same setup as the untransformed programs. We also tested manual seeds for both fuzzing variants, to increase path coverage (we refer to this as Test Augmentation - abbreviated to TA). Thus we had four set-ups: AFL, HashFuzz, AFL+TA, and HashFuzz+TA.

Lastly, we built an automated identifier extractor for patch generation (Section 3.3.3). We use *Universal Ctags* [47] (or ctags for short) to extract an initial set of variables with their types from the target files. To enrich ingredients, we exploit PyGGI's internal program representation (tagged XML) to extract expressions. We then use a custom function that infers their type. For example, assume we have the following statement: `int len = 2 * y, dist;` ctags detects `len` and `dist` and deduces that these are both integers. From this, we can infer that `2 * y` returns an integer.

4 EVALUATION

Our aim is to provide an effective automated tool that can estimate and reduce information leaks in real-world software. We therefore ask three research questions to evaluate LeakReducer. Our first question focuses on detection of information leakage:

RQ1: How effective is LeakReducer at detecting information flow leakage in a given program?

We compare LeakReducer's leak detection algorithm against a state-of-the-art fuzzing tool, using different test seeding strategies.

Our next RQ focuses on comparison with HyperGI:

RQ2: How effective is LeakReducer at reducing information flow leakage using a single-objective algorithm?

Even if LeakReducer is successful at reducing leakage using single-objective search, we still need to answer our key hypothesis: that we need to balance leakage and functionality. Thus we ask:

RQ3: How effective is LeakReducer at reducing information flow leakage using a multi-objective algorithm?

We use multiple multi-objective algorithms and examine the results from a qualitative perspective, to see if several viable solutions are found on the Pareto front.

4.1 Benchmarks

We use six programs for our study, with associated security policies. We present these in Table 1. The first three programs are the same that were used in Mesecan et al. [31]. Two of these have been used in prior research on information leakage [22] and have been reported in the Common Vulnerabilities and Exposures (CVE) database [41].

The *classify* program is similar to the *triangle* program, in that it outputs a class depending on the input parameters. In contrast to the *triangle* program, however, the input space is further divided - rather than 3 possible outputs (scalene, isosceles and equilateral) in *triangle*, there are 11 in *classify*. We introduced this program to specifically show tradeoffs between leakage and functionality loss.

The last two programs are real programs meant to show the scalability and practicability of LeakReducer. Both are taken from the OpenSSL library [42, 43]. The first is: `dtls1_process_heartbeat` function from `openssl-1.0.1f` containing the Heartbleed bug.

For our second program, we chose another OpenSSL library, but one that has no known leaks. We first examined functions within the OpenSSL crypto directory, and then looked for functions which possibly have interaction between low and high security information where a security policy might be needed. We looked for a function that had a similar signature to Hearbleed to ensure we could create a realistic security policy. In the same version of OpenSSL we found the `BIGNUM *BN_bin2bn()` function in the *bn_lib.c* file of the BigNum library. This function is one of the data entry functions to the library and has the possibility of facilitating interaction between high and low privileged users' information. Given the similarity between Heartbleed and BigNum, security policy creation was straightforward (see Table 1). Although, we used `openssl-1.1.1j` [44] in our tests, the same function is still in use in up-to-date OpenSSL versions.

4.2 Experimental Protocol

Next, we describe the methodology we use to answer our RQs.

4.2.1 Automated Test Generation. To answer RQ1 about LeakReducer's effectiveness in finding information leaks, we automatically generate two types of tests: functional (Section 3.1) and hypertests (Section 3.2). Although we use functional tests primarily to establish intended program behaviour, such tests might also reveal information leakage. In particular, we use AFL fuzzer's address sanitizer [26] option, which allows detection of subtle memory access issues (such as out-of-bounds or use-after-free).

We use four different Fuzzer-Test Setups (FTS): 1) AFL 2) AFL-TA 3) HashFuzz and 4) HashFuzz-TA, as discussed in Section 3.1. For the Test Augmentation (TA) phases we use up to 2 manual seeds, to reach the hard to find paths that fuzzing may miss. We first run each fuzzer-test setup 5 times with the AFL_EXIT_WHEN_ DONE option. With this option set, AFL automatically terminates when all discovered paths have been fuzzed many times without any new path being found [2]. From these first 5-runs, we identified the *max fuzz-time*. Then, to maximize path coverage, we run each fuzzer-test setup 5 times again for *max fuzz-time* seconds.

To generate hypertests we use the algorithm described in Section 3.2. Initially we use a budget of 250 tests. If we do not find leakage, we double the budget. Fortunately for our subjects we only needed to increase the budget once, for the Triangle program.

For each of the test sets we report on the number of unique test cases produced, generation time budgets, crashes found (whether they reveal leaks or not), as well as QIF values for hypertests.

4.2.2 Parameter Tuning. Since genetic programming has been the dominant search strategy in GI [36], we use it in LeakReducer's single-objective setting. However, it's not obvious which parameter settings would be optimal for our particular application domain. Therefore, we conduct parameter tuning on the four smaller subjects, (i.e., Triangle, Classify, Atalk and Underflow) before running the information leakage reduction stage of LeakReducer. In particular, we vary: population size — between 20 and 100, in 20 increments, as these are the min and max found in the GI literature — and mutation and crossover rates — each tried with four values: 0.25, 0.5, 0.75 and 1. We use a budget of $2,000$ program variant evaluations for each setting. Moreover, we repeat each setup 5 times, to account for the heuristic nature of GP. Overall, we test $5 * 4 * 4 = 80$ configurations of single-objective LeakReducer for each of the 4 test-fuzzer setups for each of the 4 test subjects, repeated 5 times each, resulting in a total of $6,400$ individual runs.

Recall that we use functional test fail rate and QIF from hypertesting to calculate the fitness value of each program variant. Thus, in order to fairly compare runs which use different functional test sets, we combine all 4 test sets for evaluation of the best individual found in each run. This is not an issue for hypertests, which are generated once per subject. We use the ANOVA test [45] on fitness values to test whether there is a difference in means among all configurations or not. We also test the difference in means for each configuration using the Student's t-test [46].

4.2.3 Leak Reduction. To answer RQ2 and RQ3 about LeakReducer's effectiveness at leak reduction, we run LeakReducer's GI stage using two search strategies: single- and multi-objective. We set the GI budget to $10,000$ evaluations. We repeat each repair run 20 times. As before, the fail rate in the reported final fitness calculation is based on runs of the combined set of all functional tests.

Single-Objective Optimization We use the best parameter settings from the tuning stage for each of the functional test sets for each of our subject programs. We record the fitness values, runtimes, as well as functional test fail rates and QIF values. We also re-run evolved individuals from the HyperGI work (thanks to the authors for releasing those to us).

Multi-Objective Optimization We selected four multi-objective optimization algorithms: NSGAII, NSGAIII, MoCell and SPEA2. We selected the first three, as they showed good performance in recent work [40]. We added SPAE2, since it proved successful in improvement of non-functional properties of software using genetic improvement [50]. Based on Li et al. [25]'s guidance, we report the

Table 1: Study subjects. For each we provide: a reference; the lines of code in the file containing the function of interest (which LeakReducer targets); a CVE number if information leakage was reported for this function; and the security policy used, with parameters from the function's signature and *function return values.*

				Security Policy		
Subject	**Ref**	**LoC**	**CVE-#**	High input	Low input	Low Output
Triangle	[31]	14	–	secret	side2 & side3	*function return value*
Atalk	[22]	33	CVE-2009-3002	–	sock & peer	*function return value* & uaddr
Underflow	[22]	18	CVE-2007-2875	h	ppos	*function return value*
Classify	authors	18	–	high	low	*function return value*
Heartbleed	[42]	1,082	CVE-2014-0160	–	payload_sent & payload_length	payload_received
Bignum	[43]	778	–	–	s, len & ret	s & *function return value*

following quality indicators of the Pareto fronts generated by the selected algorithms: Hyper-volume, Inverted Generational Distance, Epsilon and Generational Distance.

4.3 Test Environment

Due to compatibility issues we ran the Heartbleed fuzzer experiments on a single core of an Intel (R) Core(TM) i5-2500 CPU @ 3.3GHz processor, with 8GB RAM and 500GB HDD with Ubuntu 20.04 and gcc & g++ 7.3.0. All other experiments were run on a High Performance Computing (HPC) cluster using single cores of an Intel(R) Xeon(R) Gold 6244 CPU @ 3.60GHz and 8GB of RAM, in RedHat Enterprise Linux 7 with gcc & g++ 10.2.0.

4.4 Threats to Validity

We present the primary threats to validity of our study here. First, with respect to generality, we experimented on a limited number of programs, some with similar characteristics. However, (1) we wanted to compare against the state-of-the-art and (2) security policies, which are necessary for detecting information flow leakage cannot be developed without extensive knowledge of a system. Hence we focused on programs that have only two different classes of policy. With respect to internal validity, we acknowledge there could be faults in our programs, but we have manually validated our patches, and are providing them on our online website along with our other artifacts. We also contacted developers of the program for which LeakReducer found a possibly new information leakage fault. Last, with respect to construct validity we could have chosen different metrics, but we have used the most common metrics for evaluating multi-objective optimization and use the same measure of leakage as prior authors.

5 RESULTS

In this section we answer each of our research questions. Artifacts for the experiments are found on our anonymous website[1].

5.1 Parameter Tuning Results

Our parameter tuning showed that runs of single-objective LeakReducer that produced the fittest program variants had the following configuration: population size = 100, mutation rate = 1, and crossover rate = 0.5. We thus use these parameters in the following experiments. All our data is available on our anonymous website.

5.2 RQ1: Leak Detection

To answer this research question, we examine the data in Table 2. All of the fuzzing variants found crashes for Heartbleed and Bignum. We examined the crashes, and indeed they were caused by memory issues that lead to leakage. As the Bignum crash was not from a confirmed bug, we reached out to the developers who confirmed our suspicion that this particular function is typically not accessible from outside of a program and in the case that it is, it would be the responsibility of application developers to ensure proper use of the function. However, the code has similarities to that found in many other confirmed cases of information leakage in the wild. With an appropriately constructed wrapper to read low input and output the function result to low output, Bignum provided an interesting demonstration of the trade-off between reducing information leakage and retaining original functionality.

For the other four programs no crashes were produced by fuzzing, even though the time budget given for each program was significantly higher than the time needed to generate hypertests. Furthermore, running LeakReducer's hypertests revealed leakage in all 6 subjects, producing non-zero QIF values in all runs.

Answer to RQ1 (Leak Detection): LeakReducer is able to detect and estimate leakage in all 6 subjects, where state-of-the-art fuzzers only find leakage-related crashes in 2 of the subjects.

5.3 RQ2: Single-objective Leak Reduction

To answer RQ2 we turn to Table 3. This table shows the post patch (i.e., improved) QIF (QIF_i), Fail Rate (FR) and Fitness (F) for HyperGI and single-objective LeakReducer with 4 different inputs from fuzzing. To compare HyperGI and LeakReducer's results we ran each evolved program using our test cases (all 4 functional test sets + hypertests), for fair comparison of fitness values. The best fitness values are highlighted in bold. Although AFL-TA provides slightly better median fitness for 5 out of 6 test subjects, ANOVA and t-test on the means do not report this result as statistically significant.

Interestingly, LeakReducer did not find the optimal solution for Atalk, unlike HyperGI. Through a further communication with Mesecan et al., we found out that during identifier detection stage (see Section 3.3.3) they added an identifier declared in a header file.

Table 2: Leak detection and functional test case generation using four variants of fuzzing compared against LeakReducer's Hypertest set. For each of the fuzzing settings we show if the system crashed. For LeakReducer we show the original QIF_0. We also give the test suite size (TS), max fuzzing time budget and hypertest generation runtime in hours.

	AFL			AFL-TA			HashFuzz			HashFuzz-TA			Hypertests		
Subject	Crashes	TS	Hours	Crashes	TS	Hours	Crashes	TS	Hours	Crashes	TS	Hours	QIF_0	TS	Hours
Atalk	No	12	1.0	No	18	0.9	No	122	16.2	No	116	12.9	1.1	50	1.0
Bignum	Yes	3	7.8	Yes	24	6.4	Yes	12	21.2	Yes	70	34.7	2.4	40	0.1
Classify	No	107	2.6	No	122	5.7	No	178	8.5	No	176	8.1	2.4	125	0.2
Heartbleed	Yes	47	27.8	Yes	60	76.1	Yes	84	266.1	Yes	109	145.4	2.3	40	0.4
Triangle	No	60	3.1	No	54	1.7	No	66	58.3	No	78	69.2	0.8	194	0.1
Underflow	No	20	0.4	No	40	0.5	No	234	13.4	No	252	16.2	5.4	100	1.3

Table 3: Fitness values for best evolved variants for HyperGI and single-objective LeakReducer, guided by 4 fuzzer test sets. Medians of 20 runs are shown. In addition, we report the two fitness components: fail rates (FR) and leakage estimates (QIF_i).

				LeakReducer											
	HyperGI			AFL			AFL-TA			HashFuzz			HashFuzz-TA		
Subject	QIF_i	FR	Fitness	QIF_i	FR	Fitness	QIF_i	FR	Fitness	QIF_i	FR	Fitness	QIF_i	FR	Fitness
Atalk	0.00	0.00	**0.00**	1.00	0.00	**0.50**	0.94	0.24	**0.50**	1.00	0.00	**0.50**	1.00	0.00	**0.50**
Triangle	0.00	0.69	0.34	0.23	0.23	0.24	0.15	0.38	0.23	0.10	0.30	**0.22**	0.11	0.29	0.23
Underflow	0.00	0.62	**0.31**	0.11	0.57	0.35	0.11	0.56	**0.31**	0.64	0.01	0.37	1.00	0.00	0.50
Bignum	-	-	-	0.00	0.78	0.39	0.00	0.60	**0.30**	0.00	0.78	0.39	0.33	0.47	0.39
Classify	-	-	-	0.00	0.64	0.32	0.02	0.61	**0.31**	0.00	0.64	0.32	0.00	0.63	**0.31**
Heartbleed	-	-	-	0.50	0.00	**0.25**	0.50	0.00	**0.25**	0.50	0.00	0.43	0.50	0.00	**0.25**

That identifier was not used in the file with the target function, thus LeakReducer had no access to it. As a result a solution that fully reduces leakage was not found. We argue, however, the automation of the process outweighs potentially missing such fixes. LeakReducer still found some program variants that reduced leakage. These partial fixes could still be helpful for the developers for understanding and fixing such leaks. Moreover, the ingredient space can be enlarged in the future.

Answer to RQ2 (Single-objective Leak Reduction): Single-objective LeakReducer achieves results competitive with HyperGI, but without the need for human interaction during the repair process. Search for repairs guided by tests from AFL with seeded inputs did produce marginally better results than other settings, but the result was not statistically significant.

5.4 RQ3: Multi-objective Leak Reduction

To compare different multi-objective variants we used 4 quality indicators, as previously discussed. Since we did not see a clear winner for the functional test sets and to ensure high coverage, we simply use all functional tests generated from the previous steps, and use those during search to evaluate correctness of the evolved program variants. To compare Pareto fronts from different runs, we first prepared the *global Pareto front*, i.e., Pareto front that combines all generated fronts. Next, we calculated the hypervolume and distance from the global Pareto front to individual fronts. Table 4 reports median distances and hypervolume for each subject and algorithm. Counting the number of times each algorithm produces the best result (largest Hypervolume or Minimum Median Distance to the global Pareto front for the other metrics), we get: (1) SPEA2: 19; (2) NSGA-III: 15; (3) NSGA-II: 11; and (4) MOCell: 8. These results indicate that the SPEA2 option performs best in this problem domain. Consequently we will look more closely at the Pareto fronts generated by LeakReducer using SPEA2.

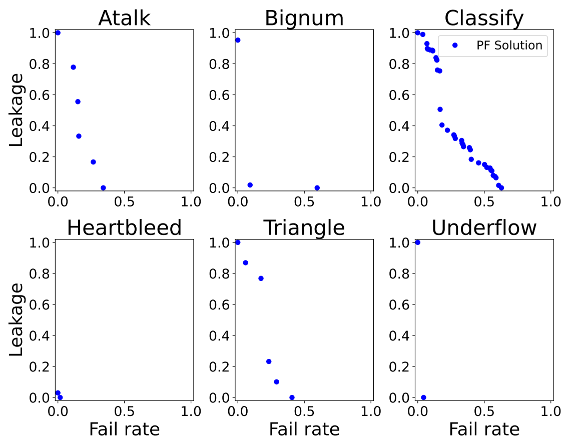

Figure 2: Sample Pareto fronts from LeakReducer with the SPEA2 setting. Each dot represents a solution on the front.

Figures 2 and 3 show results of individual and all runs of LeakReducer with the SPEA2 setting, with the best solution from single-objective LeakReducer marked. Interestingly, the single-objective

Table 4: Pareto front quality indicators per subject & multi-objective algorithm. Median of 20 runs reported. A larger hypervolume indicates a better result. For other indicators smaller numbers are closer to the global front, hence better.

	Inv. Generational Distance				Generational Distance				Hyper Volume				Epsilon			
	MOCell	NSGAII	NSGAIII	SPEA2	MOCell	NSGAII	NSGAIII	SPEA2	MOCell	NSGAII	NSGAIII	SPEA2	MOCell	NSGAII	NSGAIII	SPEA2
Atalk	0.15	0.43	0.15	**0.14**	0.04	**0.03**	0.12	0.11	0.67	0.23	0.71	**0.72**	0.24	0.73	0.22	**0.20**
Bignum	**0.38**	**0.38**	**0.38**	**0.38**	**0.33**	0.37	0.38	0.35	0.46	0.43	0.43	**0.49**	**0.40**	**0.40**	**0.40**	**0.40**
Classify	0.08	0.08	**0.07**	**0.07**	0.08	0.08	**0.07**	**0.07**	0.68	0.67	**0.69**	**0.69**	0.13	**0.12**	0.13	**0.12**
Heartbleed	**0.02**	**0.02**	**0.02**	**0.02**	0.50	0.50	0.50	**0.49**	**0.97**	**0.97**	**0.97**	**0.97**	**0.03**	**0.03**	**0.03**	**0.03**
Triangle	0.04	**0.03**	**0.03**	**0.03**	0.03	0.02	0.02	**0.01**	0.75	**0.76**	**0.76**	**0.76**	0.09	**0.05**	**0.05**	**0.05**
Underflow	**0.20**	0.30	**0.20**	0.58	0.24	**0.00**	**0.00**	**0.00**	0.66	0.54	**0.67**	0.00	**0.28**	0.43	**0.28**	1.00

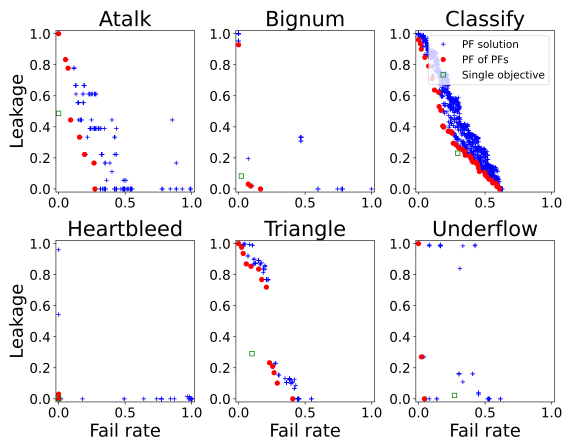

Figure 3: Pareto fronts from all 20 LeakReducer runs with the SPEA2 setting (blue), and Pareto front of all those 20 fronts (red). The green square shows the best solutions found by single-objective LeakReducer.

Table 5: Means and standard deviations of the numbers of solutions in Pareto fronts of multi-objective LeakReducer.

	MOCell		NSGAII		NSGAIII		SPEA2	
	Mean	Stdev	Mean	Stdev	Mean	Stdev	Mean	Stdev
Atalk	5.4	1.4	5.9	1.4	5.5	1.5	**6.7**	2.2
Bignum	**3.0**	0.0	**3.0**	0.3	**3.0**	0.2	**3.0**	0.0
Classify	**35.8**	4.3	33.8	3.7	31.9	4.1	31.7	5.1
Heartbleed	**2.1**	0.2	**2.1**	0.2	2.0	0.0	2.0	0.0
Triangle	6.9	1.8	7.3	1.5	**7.5**	1.5	7.4	1.6
Underflow	3.5	1.1	3.5	1.2	3.5	1.2	**4.1**	1.3
Average	9.42	-	9.24	-	8.88	-	9.13	-

variant produces results that are on or close to multi-objective LeakReducer's Pareto fronts. However, the latter option provides more variants to choose from, leveraging leakage vs fail rate.

We are also interested in the diversity of solutions found. Table 5 presents the average number of solutions found and the standard deviations out of 20 runs for each algorithm and subject pair. Firstly, the average number of solutions in Pareto fronts is quite close; ranging from 8.9 for NSGAIII to 9.4 for MOCell. Secondly, MOCell and SPEA2 generated the largest number of solutions for 3 subjects, while NSGAII and NSGAIII generated the largest number of solutions for 2 subjects. We will discuss the quality of solutions in the next section. Taking all results into account, however, we would recommend the use of SPEA2 with LeakReducer, as it is likely to produce a diverse set of solutions, close to the optimal Pareto front.

Answer to RQ3 (Multi-Objective LeakReducer): Both single- and multi-objective settings produced effective fixes, with single-objective LeakReducer producing solutions close to the multi-objective LeakReducer's Pareto fronts, while the latter provided a diverse set of solutions balancing leakage and fail rate. Moreover, we recommend the use of SPEA2 with LeakReducer, as we showed with multiple indicators that it generally outperforms other multi-objective algorithms.

5.5 Discussion

Atalk is the real-world program that showed the most evenly spread Pareto front in Figure 2. The original leaky function is:

```
struct atalk_sock {
  unsigned char dst_node, src_node, dst_port, src_port;
  int sk_state;
  char res[16];
};
```

```
int atalk_getname(atalk_sock *sock, atalk_sock *uaddr, int peer) {
  struct atalk_sock sat;
  int err = -ENOBUF;
  if (sock_flag(sock))
    goto out;

  if (peer) {
    err = -ENOTCON;
    if (sock->sk_state != TCP_ESTABLISHED)
      goto out;
    sat.src_node = sock->dst_node;
    sat.src_port = sock->dst_port;
    sat.dst_node = sock->src_node;
    sat.dst_port = sock->src_port;
  } else {
    sat.src_node = sock->src_node;
    sat.src_port = sock->src_port;
    sat.dst_node = sock->dst_node;
    sat.dst_port = sock->dst_port;
  }

  sat.sk_state = sock->sk_state;
  memcpy(uaddr, &sat, sizeof(sat));
  err = sizeof(atalk_sock);

out:
  return err;
}
```

This particular program leaks values from internal memory due to the struct `sat` defined on line 2 being uninitialised. There are 6 struct members, but only 5 are assigned to in the code in lines 3-22. The entire memory contents of the struct are then copied to the function parameter `uaddr`, including the value of the uninitialised 6th struct member '`res`'. As `sat` is a local variable, this uninitialised memory contains stack data, which could, depending on previous function call stacks, contain sensitive data. For the sake of brevity, we show 2 of the best patches generated by SPEA2:

Patch 2; Leakage: 0.000, Functional: 0.341

```
10a11
>       sat.src_port = sock->src_port;
18a20,22
>       if (sock->src_port < sock->src_node) {
>         sock->src_node = sock->src_port;
>       }
20a25,27
>     if (sock->dst_port != sock->src_node) {
>       return 1;
>     }
23a31,33
>     for (int lcv1476 = 0; lcv1476 < peer; lcv1476++) {
>       err = -ENOTCON;
>     }
```

Patch 5; Leakage: 0.778, Functional: 0.116

```
6a7
>     err = -ENOTCON;
10a12,14
>       for (int lcv943 = 0; lcv943 < TCP_ESTABLISHED; lcv943++) {
>         sat.src_port = sock->src_port;
>       }
21a26,28
>     if (sizeof(sat) > sock->src_port) {
>       return 0;
>     }
```

There is variation in these patches, and they are both returning a fixed value before the leak occurs (in line 23) depending on a comparison between structs `sat` and `sock`. The variations in leakage and functional test performance can be attributed to different comparisons between the two structs. These comparisons cause an early return before the leakage occurs in differing proportions of the test inputs, hence differing functional results and leakage rates.

An ideal patch would initialize the `sat.res` struct member to a fixed value. As this variable is in fact an array, this would require a `for` loop, or a call to `memset`. As an alternative, using the short form compound literal initialiser `struct atalk_sock sat = {0};` will initialize all struct members (including all array members) to all zeroes. Neither of these are used in the existing code, so the GI would require additional mutation operators to produce these.

The 3 other real-world programs (Heartbleed, Bignum and Underflow) all contain developer bugs, which leads to a less diverse range of solutions. This is because they can in theory all be 'fixed' in a way that retains all original functionality, whilst completely eliminating the leakage. Both Classify and Triangle have leakage that is caused by intentionally poor information flow control design, as such these produce a broad, dense Pareto front showcasing a variety of potential solutions.

The best single-objective patch found for Heartbleed was as follows (red code was removed in the patch, and green code was added by the patch):

```
int dtls1_process_heartbeat(SSL *s) {
  ...
  unsigned int payload, padding = 16;
  n2s(p, payload);  /* Read payload length */
  ...
  unsigned char *buffer = OPENSSL_malloc(1 + 2 + payload + padding);
  unsigned char *bp = buffer;

  *bp++ = TLS1_HB_RESPONSE;  /* Copy response type into buffer */
  s2n(payload, bp);          /* Copy payload length into buffer */
  memcpy(bp, pl, payload);   /* Copy payload into buffer */
  bp += payload;             /* Update buffer pointer */
  if (SSL_F_DTLS1_PREPROCESS_FRAGMENT < 1 + 2 + payload + padding) {
    return 0;
  }
  RAND_pseudo_bytes(bp, padding);

  /* Send callback */
  r = dtls1_write_bytes(s, TLS1_RT_HEARTBEAT, buffer, 3 + payload + padding);
  ...
```

The added `if`-statement fixes many leaks because of the way that Heartbleed is exploitable. Essentially the `payload` length in the malformed packet is set to some value larger than the length of the actual message sent to be echoed back. `padding` is always 16. The best multi-objective search results are semantically equivalent patches to this. The payload length is read in by a function-like macro `s2n`, which reads an unsigned 2-byte (16-bit) integer (between 0-65535) from the transmitted heartbeat buffer. The leak occurs when `payload` bytes are copied into the callback message and sent back to the other client; thus in order to maximise leakage, an attacker should set this value to 65535 (the maximum) which will result in 65535 - 19 - `ACTUAL_PAYLOAD_LENGTH` bytes of internal program memory being returned to them in the corresponding heartbeat response. The `-19` comes from the 16 bytes of `padding` and an extra 3 bytes for indicating message type and payload length. The constant `SSL_F_DTLS1_PREPROCESS_FRAGMENT` is defined as 288 in a header file. This patch is therefore discarding any hearbeat request where `payload` is larger than 269 ($288 - 19$), which reduces the leakage from a potential 65516 bytes down to 269 bytes. As described above, an attacker looking to maximise leakage would be setting `payload` to very large values, and this patch would silently discard these. The fuzzer however is not using any malicious heuristics when generating malformed packets, but instead generates the buffer (containing the payload length) pseudo-randomly, and with the leaking search space shrinking down to 0.41% (269/65535) of what it originally was, the leak is seemingly repaired. It is worth noting that the patch will also discard any properly formed heartbeat packets with an actual payload of length 270 or greater.

The developer patch is quite close to this, in fact, the only thing different is that the comparison value `SSL_F_DTLS1_PREPROCESS_FRAGMENT` is replaced with the actual length of the received buffer. This eliminates the leakage for requests with indicated payloads <= 269 bytes, whilst accepting properly formed packets with an actual payload length > 269 bytes. An improved test set would expose the remaining (much smaller) leak, and is a target for future research.

The evaluation criteria does well to guide the repair process, as is evidenced by the proposed patch being very close to the actual developer patch. An acclimatised OpenSSL developer would see that comparing the variable payload length with a constant value would create issues for properly formed long heartbeat requests, but learn that the leakage is strongly correlated to large payload values. It is not a leap to suggest that a developer could come to the correct conclusion that the solution is to discard requests where the actual buffer length does not match the indicated buffer length.

In the case of Atalk, we already discussed the issues with the generated patches. Bignum leaks information via a buffer-overflow, and does not have a single 'fix', however we do see patches that manage to greatly reduce the quantity of leakage while not resulting in a large functionality testcase fail rate increase. The best patches generated through multi-objective LeakReducer for Underflow reduce leakage to a greater extent than the best single objective patch, whilst simultaneously retaining greater functionality.

6 CONCLUSIONS AND FUTURE WORK

We have presented LeakReducer, a multi-objective framework to detect, localize and repair information leakage in real-world programs. We have evaluated LeakReducer on a set of programs and were able to find leaks in all of them. We demonstrated our ability to repair the leaks against repairs from the state-of-the-art tool, HyperGI. For those programs whose leakage and functionality are competing, the multi-objective setting provided a diverse Pareto front which can be used to balance leakage and functionality. We plan to add additional repair ingredients, more specialised mutation operators, refined identifier resolution, and apply LeakReducer to a larger set of programs and different security policies.

ACKNOWLEDGMENTS

This work is supported in part by NSF grants CCF-1909688, CCF-1901543 and by EPSRC grant no. EP/P023991/1.

REFERENCES

[1] 2022. https://github.com/anonymous183459/LeakReducer/.

[2] AFL 2022. American Fuzzy Lop plus plus (AFL++). https://github.com/AFLplusplus/AFLplusplus. Accessed: 2022-05-22.

[3] Johan Agat. 2020. Transforming Out Timing Leaks. In *POPL 2000, Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Boston, Massachusetts, USA, January 19-21, 2000*, Mark N. Wegman and Thomas W. Reps (Eds.). ACM.

[4] M. S. Alvim, K. Chatzikokolakis, A. McIver, C. Morgan, C. Palamidessi, and G. Smith. 2020. *The Science of Quantitative Information Flow*. Springer.

[5] Gabin An, Aymeric Blot, Justyna Petke, and Shin Yoo. 2019. PyGGI 2.0: Language Independent Genetic Improvement Framework. In *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 1100–1104.

[6] Andrea Arcuri. 2019. RESTful API automated test case generation with EvoMaster. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 28, 1 (2019), 1–37.

[7] Owen Arden, Jed Liu, Tom Magrino, and Andrew Myers. 2016. Jif: Java with Information Flow. https://www.cs.cornell.edu/jif/. Accessed: 2022-05-02.

[8] Konstantinos Athanasiou, Thomas Wahl, A. Adam Ding, and Yunsi Fei. 2020. Automatic Detection and Repair of Transition- Based Leakage in Software Binaries. In *Software Verification - 12th International Conference, VSTTE 2020, and 13th International Workshop, NSV 2020, July 20-21, 2020, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 12549)*, Maria Christakis, Nadia Polikarpova, Parasara Sridhar Duggirala, and Peter Schrammel (Eds.). Springer, 50–67.

[9] Antonio Benitez-Hidalgo, Antonio J Nebro, Jose Garcia-Nieto, Izaskun Oregi, and Javier Del Ser. 2019. jMetalPy: A Python framework for multi-objective optimization with metaheuristics. *Swarm and Evolutionary Computation* 51 (2019), 100598.

[10] Fabrizio Biondi, Michael A. Enescu, Annelie Heuser, Axel Legay, Kuldeep S. Meel, and Jean Quilbeuf. 2018. Scalable Approximation of Quantitative Information Flow in Programs. In *Verification, Model Checking, and Abstract Interpretation - 19th International Conference, VMCAI*, Isil Dillig and Jens Palsberg (Eds.). Springer.

[11] Niklas Broberg, Bart van Delft, and David Sands. 2017. Paragon - Practical programming with information flow control. *Journal of Computer Security* 25, 4-5 (2017), 323–365.

[12] David Clark, Sebastian Hunt, and Pasquale Malacaria. 2001. Quantitative Analysis of the Leakage of Confidential Data. *Electronic Notes in Theoretical Computer Science* 59, 3 (2001), 238–251.

[13] David Clark, Sebastian Hunt, and Pasquale Malacaria. 2007. A static analysis for quantifying information flow in a simple imperative language. *Journal of Computer Security* 15, 3 (2007), 321–371.

[14] Michael R. Clarkson and Fred B. Schneider. 2010. Hyperproperties. *Journal of Computer Security* 18, 6 (2010), 1157–1210.

[15] Jared L Cohon. 2004. *Multiobjective programming and planning*. Vol. 140. Courier Corporation.

[16] Thomas M. Cover and Joy A. Thomas. 2006. *Elements of Information Theory (Wiley Series in Telecommunications and Signal Processing)*. Wiley-Interscience.

[17] Dorothy E. Denning. 1976. A Lattice Model of Secure Information Flow. *Commun. ACM* 19, 5 (1976), 236–243.

[18] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++: Combining Incremental Steps of Fuzzing Research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association.

[19] Gordon Fraser and Andrea Arcuri. 2011. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 416–419.

[20] Joseph A. Goguen and José Meseguer. 1982. Security Policies and Security Models. In *1982 IEEE Symposium on Security and Privacy*. 11–20.

[21] S. He, M. Emmi, and G. Ciocarlie. 2020. ct-fuzz: Fuzzing for Timing Leaks. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE Computer Society.

[22] Jonathan Heusser and Pasquale Malacaria. 2010. Quantifying information leaks in software. In *Twenty-Sixth Annual Computer Security Applications Conference, ACSAC*. 261–269.

[23] Pieter Hooimeijer, Martino Luca, Peter O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. 2015. Moving Fast with Software Verification. In *NASA Formal Methods: 7th International Symposium, Proceedings*, Vol. 9058. Springer, 3.

[24] Johannes Kinder. 2015. Hypertesting: The Case for Automated Testing of Hyperproperties. In *Hot Issues in security and trust (HotSpot)*.

[25] Miqing Li, Tao Chen, and Xin Yao. 2020. How to Evaluate Solutions in Pareto-based Search-Based Software Engineering? A Critical Review and Methodological Guidance. *IEEE Transactions on Software Engineering* 01 (2020), 1–1.

[26] LLVM Foundation 2022. Clang 15.0.0git documentation, AddressSanitizer. https://clang.llvm.org/docs/AddressSanitizer.html. Accessed: 2022-05-02.

[27] Pasquale Malacaria, Michael Tautchning, and Dino Distefano. 2016. Information Leakage Analysis of Complex C Code and Its application to OpenSSL. In *Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques - 7th International Symposium, ISoLA 2016, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 9952)*. 909–925.

[28] Valentin JM Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. 2019. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering* 47, 11 (2019), 2312–2331.

[29] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th International Conference on Software Engineering (ICSE)*. 691–701.

[30] Hector D Menendez and David Clark. 2021. Hashing fuzzing: introducing input diversity to improve crash detection. *IEEE Transactions on Software Engineering* (2021).

[31] Ibrahim Mesecan, Daniel Blackwell, David Clark, Myra B Cohen, and Justyna Petke. 2021. HyperGI: Automated Detection and Repair of Information Flow Leakage. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1358–1362.

[32] Jonathan Metzman, László Szekeres, Laurent Simon, Read Sprabery, and Abhishek Arya. 2021. FuzzBench: An Open Fuzzer Benchmarking Platform and Service. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 1393–1403. https://doi.org/10.1145/3468264.3473932

[33] Jonathan Metzman, László Szekeres, Laurent Simon, Read Sprabery, and Abhishek Arya. 2021. FuzzBench: an open fuzzer benchmarking platform and service. In *ESEC/FSE*. 1393–1403.

[34] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. Semfix: Program repair via semantic analysis. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 772–781.

[35] Yannic Noller and Saeid Tizpaz-Niari. 2021. QFuzz: Quantitative Fuzzing for Side Channels. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2021)*. ACM.

[36] Justyna Petke, Saemundur O Haraldsson, Mark Harman, William B Langdon, David R White, and John R Woodward. 2017. Genetic improvement of software: a comprehensive survey. *IEEE Transactions on Evolutionary Computation* 22, 3 (2017), 415–432.

[37] Willard Rafnsson and Andrei Sabelfeld. 2016. Secure multi-execution: Fine-grained, declassification-aware, and transparent. *Journal of Computer Security* 24, 1 (2016), 39–90.

[38] Andrei Sabelfeld and Andrew C. Myers. 2003. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications* 21, 1 (2003), 5–19.

[39] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. 2018. Lessons from building static analysis tools at google. *Commun. ACM* 61, 4 (2018), 58–66.

[40] Vali Tawosi, Federica Sarro, Alessio Petrozziello, and Mark Harman. 2021. Multi-objective software effort estimation: A replication study. *IEEE Transactions on Software Engineering* (2021).

[41] The MITRE Corporation 2022. CVE – Common Vulnerabilities and Exposures. https://cve.mitre.org/. Accessed: 2022-05-02.

[42] The National Institute of Standards and Technology 2014. CVE-2014-0160 Heartbleed. https://nvd.nist.gov/vuln/detail/CVE-2014-0160. Accessed: 2022-05-02.

[43] The OpenSSL 2021. bn - multiprecision integer arithmetics. https://www.openssl.org/docs/man1.0.2/man3/bn.html. Accessed: 2022-03-12.

[44] The OpenSSL 2021. OpenSSL 1.1.1 versions download. https://ftp.openssl.org/source/old/1.1.1/. Accessed: 2022-03-18.

[45] The R Project 2022. Anova: Anova Tables for Various Statistical Models. https://www.rdocumentation.org/packages/car/versions/3.0-12/topics/Anova. Accessed: 2022-04-18.

[46] The R Project 2022. t.test: Student's t-Test. https://www.rdocumentation.org/packages/stats/versions/3.6.2/topics/t.test. Accessed: 2022-04-18.

[47] Universal Ctags Team 2021. Universal Ctags. https://ctags.io/. Last Accessed: 2022-05-02.

[48] Dennis M. Volpano, Cynthia E. Irvine, and Geoffrey Smith. 1996. A Sound Type System for Secure Flow Analysis. *Journal of Computer Security* 4, 2/3 (1996), 167–188.

[49] Shuang Wang and Jeff Offutt. 2009. Comparison of unit-level automated test generation tools. In *2009 International Conference on Software Testing, Verification, and Validation Workshops*. IEEE, 210–219.

[50] David R White, Andrea Arcuri, and John A Clark. 2011. Evolutionary improvement of programs. *IEEE Transactions on Evolutionary Computation* 15, 4 (2011), 515–538.

[51] Hirotoshi Yasuoka and Tachio Terauchi. 2014. Quantitative information flow as safety and liveness hyperproperties. *Theoretical Computer Science* 538 (2014), 167–182.